Progress Report - Year #2
Automated Extraction of Flow Features
NASA Marshall Space Flight Center Grant #NAG8-1872

Robert Haimes
Department of Aeronautics & Astronautics
Massachusetts Institute of Technology
haimes@mit.edu

July 1, 2004

1 Introduction

Computational Fluid Dynamics (CFD) simulations are routinely performed as part of the design process of most fluid handling devices. In order to efficiently and effectively use the results of a CFD simulation, visualization tools are often used. These tools are used in all stages of the CFD simulation including pre-processing, interim-processing, and post-processing, to interpret the results. Each of these stages requires visualization tools that allow one to examine the geometry of the device, as well as the partial or final results of the simulation. An engineer will typically generate a series of contour and vector plots to better understand the physics of how the fluid is interacting with the physical device. Of particular interest are detecting features such as shocks, re-circulation zones, and vortices (which will highlight areas of stress and loss). As the demand for CFD analyses continues to increase the need for automated feature extraction capabilities has become vital.

In the past, feature extraction and identification were interesting concepts, but not required in understanding the physics of a steady flow field. This is because the results of the more traditional tools like; iso-surface, cuts and streamlines, were more interactive and easily abstracted so they could be represented to the investigator. These tools worked and properly conveyed the collected information at the expense of a great deal of interaction. For unsteady flow-fields, the investigator does not have the luxury of spending time scanning only one "snap-shot" of the simulation. Automated assistance is required in pointing out areas of potential interest contained within the flow. This must not require a heavy compute burden (the visualization should not significantly slow down the solution procedure for co-processing environments). Methods must be developed to abstract the feature of interest and display it in a manner that physically makes sense.

2 A Secondary Flow Surface

2.1 Definition

The concept of secondary flow in turbomachinery is not well defined, but commonly referenced. Some attempts at rigorously defining this idea include:

- ...component of absolute vorticity in the direction of the relative streamline [Hawthorne, 1974]
- Secondary flow in broad terms means flow at right angles to the intended primary flow [Cumpsty, 1989]
- Due to viscous effects, endwalls divert primary flow produced by blades and vanes, to give rise to what has come to be called secondary flow [Bradshaw, 1996]

Of the three definitions listed above only [Cumpsty, 1989] provides a definition that could be made operational. What is required is the notion of primary flow, which we can define. Unfortunately by the time we get a CFD solution the notion of "intended" is lost.

The desire to view perturbations from the *primary* flow direction can give insight into the viscous, reverse flow and vortical effects that deviate from the design. To this end it is obviously desirable to be able to generate 2D vector plots that display the secondary flow given a traditional CFD solution.

Secondary flow plots are usually displayed in a passage between blades or just downstream from the trailing edge. The arrows are generated from a frame of reference that is relative to the passage in question (i.e. absolute for fixed rows and moving for rotors). This obviously points to a difficulty in areas between stators and rotors: what is the appropriate frame of reference?

2.2 Algorithm

It would clearly be desirable to have a scheme that could maximize the primary flow through a constructed surface. This could be done by defining a *pivot* point in the channel that reflects the some centroid of the passage or flow. A surface that goes through the point can then be generated. By adjusting the position of this surface the *best* fit can be found. This surface can then be used to view the secondary flow by projecting the vector field data onto the surface.

2.2.1 Primary Flow definition

The goal here is to calculate the mass-averaged quantities in the channel. This should be done on a grid plane or a cut through the passage that is orientated so that all bounds of the cut are walls (if possible). The following can be done with either a plane (all surface facets have the same normal n) or an analytical surface where the normal for cut facets can change.

Compute surface integrals:

$$\begin{aligned}
\bar{A} &\equiv \iint \vec{n} dA \\
\bar{M} &\equiv \iint [\rho\vec{q} \cdot \vec{n}] \, dA \\
q_0 &\equiv \bar{M}/\bar{A} \\
\vec{X}_0 &\equiv \iint \left[\vec{X}\rho\vec{q} \cdot \vec{n}\right] dA/q_0
\end{aligned}$$

where $\bar{M}$ is the mass-averaged flux, q_0 is the mean velocity and $\vec{X}_0$ is the mass-averaged center of the flow.

2.2.2 Newton-like Iteration to Maximize Primary Flow

By selecting various cuts that pass through $\vec{X}_0$ we can adjust the normal (in the case of a simple plane) in an iterative loop so that we maximize $q_{z'}$ (the velocity perpendicular to the plane):

$$\vec{n} = |q_0|$$

Note that new surface integrals are recomputed during each iteration. This will also change the position $\vec{X}_0$.

Using a planar cut this technique takes about 3 to 4 iterations to converge (i.e. the normals returned do not differ by some small factor). This Newton-like convergence is most always seen unless the planar cut is adjusted so that a new portion of the flow field is exposed.

When converged, this provides a view of the data that displays secondary flow when the normal velocity component is removed.

2.3 Discussion

In practice this algorithm works well but did require a number of operational adjustments. These included:

2.3.1 Passage of Interest

The fast cut algorithms are based on *Marching Cubes* and are performed on a 3D element at a time. The result is a set of disjoint polygons that reflect the portion of the surface that cuts through the cell. The notion of where in the domain the fragments come from is usually lost. So if the simulation contains more than a single passage the cut data can easily contain fragments from elsewhere in the simulation. This will corrupt the primary flow calculation in that we are no longer focused on a single passage.

The solution is to reconnect the fragments into complete (and bounded) surfaces. Once this is done a seed point can be located within the bounded surfaces so that one can be selected. Only those polygons that are within the selected region are used in the calculations.

The cut algorithm used constructs the surface in a Finite-Element sense (that is, a list of nodes that reflect the 3D edges being cut is constructed and the polygons refer to indices in that list). The reconnection is performed via a polygon side-matching algorithm based on the indices (not floating-point locations). This is unique and robust. Any side that is seen by two polygons is interior to the region. A side with only a single polygon is bounding the region.

2.3.2 Multi-block simulations

In multi-block simulations the volumes represented by the blocks can abut or overlap. The individual cell definitions are usually block specific so that even if the blocks maintain a larger contiguous volume, it is usually not apparent by the time one looks at the fragments from the *Marching Cubes* results. When reconnecting the regions the results will reflect the block boundaries and not the actual bounds of the cut. The regions need to be placed back together.

When performing streamlining, it is traditional to use the "IBlank" data to inform the software how the blocks are connected. When one pierces a cell on a face where the "IBlank" data indicates that a *jump* to another block is required, the "IBlank" index contains the accepting block. Initially, this data was used to attempt to flood the region from the target surface fragments to connecting blocks. This was found to be unreliable.

A much more expensive technique was developed. This involved producing a bounding-box around each region as a first step. All regions (that have not been included) and have bounding boxes that overlap the start region are examined. Each point on the exterior of the start region is compared to all fragments of the candidate regions. If it is found that any point is interior, then the new region is considered part of the calculation and this process is then recursively applied where this candidate becomes the start region.

In this way the seed point fills all connecting and overlapping areas and the calculation can proceed on that "passage".

2.3.3 Tip leakage simulations

When performing the secondary flow algorithm on a simulation that displays tip leakage there is a natural connection between passages. With the algorithm described above there will be flooding into other passages. This will corrupt the primary flow calculation.

This problem has been taken care of if the simulation is multi-block and there are individual blocks that represent the tip flow regions. The flooding can be "dammed" by informing the technique not to use certain blocks as candidates.

2.3.4 Frame of reference

In multi-stage calculations care needs to be taken so that algorithm sees data in a consistent frame of reference. This means that when looking at the secondary flow in a rotor, all velocity field values should be in the rotating frame. It is important that the data in the stators be transformed so that the technique does not see any discontinuities in the velocity field.

This then means that if one were to traverse the machine from upstream to down that there will be a number of changes of reference. These should be done while the resultant planar cut is in the zone between blades.

2.4 API addition to FX

The **F**eature e**X**traction toolkit **FX** contains an infrastructure that can handle the various different methods that a CFD solution can be discretized. This toolkit, unlike most visualization systems, is lightweight because no drawing and/or GUI functions are supported. In general, the input is the CFD solution and output is various forms of geometry.

2.4.1 FX_MeanFlow

FX_MEANFLOW(XPOS, VNORM, DAM)
This subroutine given the start position and plane normal computes the mass-averaged "center" of flow and the mass averaged velocity.

float XPOS[3]	On input the position that sets the plane given the normal VNORM. On output, the mass averaged position on the planar cut is returned.
float VNORM[3]	On input the normal that sets the family of planes to use to produce the cut. On output VNORM is filled with the mass averaged velocity through the cut.
int *DAM	Pointer to the status of each block (for multi-block cases) to act as a "dam" for the flooding procedure. Zero indicates that flooding through the block is OK, a one is the flag to NOT use this block. NOTE: may be NULL to specify no "damming".

2.5 Status

An extended abstract on this work has been submitted to the 2005 AIAA Aerospace Sciences Meeting (at Reno, NV). The disposition is not yet known.

This feature technique has also been passed on to General Electric. The people responsible for the visualization codes that both GE Aircraft Engines and GE Power Generation use are located at GE Global Research. Stuart Connell manages this effort and he has incorporated the Secondary Flow finder into NPLOT3D (their visualization "workhorse" code). The initial feedback is that this feature is useful.

3 Field Interpolation

There was an effort at MSFC to be able to accurately interpolate the data from one mesh onto another where the nodal positions do not match. This interpolation can be performed in a number of ways (this is due to the fact that finite volume/finite difference CFD does not actually define a cell-based interpolant). If the interpolation is done without some accuracy, then the solution on the target mesh may be far from converged (even if the source solution was converged and the geometries are the same). This situation becomes worse in meshes dominated by boundary layer stretching – errors in these regions are easy to generate and have a significant effect.

The interpolation routines used for streamlining and unsteady particle tracing were applied to this mesh-to-mesh problem with great success.

4 Rendering of Higher-Order Finite Elements

Numerical methods are widely used throughout academia and industry to solve physical problems when experimental data is difficult to obtain. The details of these methods can vary greatly, but they all essentially solve a set of governing equations by discretizing the domain of interest and solving an analogous formulation at the discrete points or nodes. Once a solution has been generated for these nodes, then data over the entire domain can be obtained by interpolation. The simplest way to interpolate is to assume linearity within each cell based on the vertices that support that element. There are a number of ways available to then view this data, since most visualization techniques are based on the assumption of linear interpolation. However, there are many situations in which it is advantageous to solve the discrete equations using a non-linear basis or higher order elements. This can mean using anything from the standard polynomial Lagrange basis to a scheme as complicated as a hierarchical basis or spectral elements. One obvious difficulty with using higher order numerical methods is that there is no simple way to visualize the data in its native form (since most current visualization software uses a linear basis). This renders higher order methods much less useful. Understanding of numerical results and new insight is often only possible when one can accurately visualize the massive amounts of data produced.

Accurate rendering of nonlinear data cannot be performed efficiently using only the standard OpenGL API, since all OpenGL primitives are inherently linear. Higher order data can be interpolated and rendered quite simply and quickly by utilizing the flexibility of modern graphical processing units (GPUs). In addition to rendering surfaces, one important technique used in scientific visualization is the generation planar cuts through 3D field data. This can be accomplished through a combination of selective refinement of the elements and accessing the programmable shaders inside the GPU.

4.1 Discontinuous Finite Elements

One popular group of numerical techniques, the Finite Element Methods (FEM), are particularly convenient when dealing with complex geometries or unstructured computational meshes. The FEM

simplifies the solution scheme by mapping every element in the mesh to a master *reference* element, and then scalar interpolation can be performed using shape functions as a basis.

When rendering continuous data, neighboring elements share both the location and field data of common nodes. The use of collected primitives (polytriangles, quad meshes and etc.) can speed up the display time since the support data needs to be passed along the graphics pipeline fewer times. However, the direct goal of this research was to visualize flow solutions generated using the Discontinuous Galerkin (DG) Method. As such, any scheme developed should be able to naturally handle discontinuities (at element faces) in the scalar fields being visualized. The simplest way to accomplish this is for each element to independently store data for all of its basis nodes. Even though the physical location of shared nodes is the same between neighboring elements, nodes must be respecified for each element in which they appear. The goal is to have a method that allows for easy handling of both continuous and discontinuous data with the acknowledgement that there will be some lose of the speed benefits in comparison to the use of collected primitives for continuous data.

4.2 Visualization Tools

The status of the implementation of commonly used visualization tools for higher-order elements is listed below.

4.2.1 Surface Rendering

The coloring (and lighting) of the surface patches is done in an accurate manner. What is not properly handled, at this point, are curved triangles. OpenGL only rasterizes planar fragments, therefore in order to precisely render curved patches, a method to cover the *shadow* of the patch is required. This geometric fragment is view dependent and therefore changes as the view matrix is adjusted. This portion of the algorithm has not been completed.

What has been accomplished is that a p_1, p_2 and p_3 scalar evaluators have been implemented. Unlike OpenGL where interpolation is performed in color space, here proper scalar interpolation is computed in the graphics hardware and the color applied from a colormap stored in texture memory. Once the color has been found, the same interpolation can be performed on the geometry. This can give an accurate normal on the curved patch. This normal is the one that then gets applied for the lighting calculations. Also, the depth is properly adjusted (and not taken from the linear interpolation of the fragment). This does a remarkably good job in providing a visual representation of the patch even though it is based on the linear raster positions.

4.2.2 Planar Cuts

A scheme to properly render cutplanes has been implemented. Please see the attached paper *Rendering Planar Cuts Through Quadratic and Cubic Finite Elements.* This paper has been accepted to the IEEE Visualization conference and will be presented in October 2004.

4.2.3 Iso-surfaces

Preliminary efforts have begun. The algorithms to render each type of intersection for linear elements are the same as with planar cuts. The crucial difference is that iso-surface will, in almost all cases, be guaranteed not to be planar.

However, it may be possible to render the isosurface with scalar value, s^*, by bounding it with linear primitives. Based on screen position, x_s, of each pixel on the bounding shadow, the depth is adjusted until the point on the isosurface, x, is found such that x_s lies on top of x (i.e. x and x_s have the same screen coordinates but different depths). To find x, $|s - s^*|$ is first minimized by performing a search of points inside the element that lie beneath x_s, then the fragment can rejected or drawn based on whether or not $s = s^*$. Performing this search would be relatively expensive, so acceptable values of s will lie close to s^* within some bounds set by the accuracy of the search. Under some viewing transforms, the isosurface can curve behind itself, which means there can be multiple solutions, x, that all lie on top of x_s. In this case, the several solutions should be compared using the depth test to determine which one is displayed.

How are the bounding shadow primitives determined to render the isosurface? The faces of the congruent tetrahedron used to generate the cutplane shadow would certainly cover the isosurface intersection, since it captures the entire element by design. But using those triangles could produce many extraneous fragments.

4.2.4 Steamlines

This has not been started. For continuous data higher order interpolation is not a problem. The normal streamline and unsteady particle tracer is only a function of the velocity field (at optionally its gradient) at requested points. FEM is designed to provide accurate interpolation. Routines are required for each type of element supported in the simulation.

There is a problem for DG simulations. Many of the numerical techniques used for particle/streamline integration assume continuous field data. It is not clear what will happen to the results when there are jumps seen at element boundaries. Those techniques like variable step Runge-Kutta integration will fail. In fact, the concept of a streamline in a discontinuous simulation may not be well defined.

4.3 Status

The student performing much of this effort, Michael Brasher, will be graduating with his Masters degree at the end of the summer.

to accomplish this is for each element to independently store data for all of its basis nodes, similar to [8]. Even though the physical location of shared nodes is the same between neighboring elements, nodes must be respecified for each element in which they appear. The goal is to have a method that allows for easy handling of both continuous and discontinuous data with the acknowledgement that there will be some lose of the speed benefits in comparison to the use of collected primitives for continuous data.

3.1 Reference Element Interpolation

In general, a triangular element T has a scalar interpolant of order p and q degrees of geometrical freedom. The degrees of freedom determine if and how the sides of T are curved, and the order of interpolation determines how many nodal values of the scalar function are needed to specify the interpolant. For example, a p_3q_2 triangle would have a cubic polynomial scalar interpolant and quadratic geometry.

Using the Lagrange basis, every element in the mesh can be mapped to a reference element. The reference coordinates, $\bar{\xi}$, are aligned so that the component ξ_i is 1 at vertex i of the reference element and 0 at all other vertices. Note that there are 3 reference coordinates in 2D and 4 reference coordinates in 3D. The extra degree of freedom is removed by requiring that the coordinates identically sum to 1, i.e. $\sum_i \xi_i = 1$. The nodal shape functions ϕ_i are defined so that at each node n_j:

$$\phi_i(n_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{1}$$

Given a scalar function with nodal values s_i at node n_i, the value of the scalar interpolant $s(\bar{\xi})$ at a point $\bar{\xi}$ is given by:

$$s(\bar{\xi}) = \sum_i s_i \phi_i(\bar{\xi}) \tag{2}$$

It is convenient to scale the nodal values so that the scalar interpolant is contained in $s \in [0,1]$. Once the value of the scalar interpolant is found at a point, the color at that point is defined by some arbitrary colormap. One standard choice of a colormap is the spectral colormap shown in fig. 1.

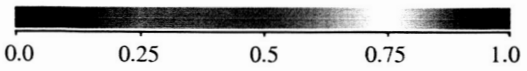

Figure 1: Spectral Colormap

In addition to nonlinear scalar data, the geometry of the element can be curved. Only the coordinates of each node in physical space, $p_i = \{x_i, y_i, z_i\}$, need to be specified, and then the geometry of the element is interpolated in the same manner as the scalar field using eq. 2. As a matter of practice in computational meshes, there will be $q > 1$ elements conforming to the curved boundaries and linear $q = 1$ elements on straight boundaries and in the interior. At times $q > 1$ interior elements may be seen when there is a stretched mesh near a curved boundary. This ensures positive volumes and well-behaved interpolation.

3.2 Dimensional Hierarchy

Given physical coordinates at the nodal points, the p_x reference elements map to some curved region in physical space, called a p_x tetrahedron in 3D, a p_x triangle in 2D, and a p_x line in 1D. The four faces of a p_x tetrahedron can be mapped to the 2D reference element, so each face can be described as a p_x triangle. Similarly, the three edges of a p_x triangle can be described as a p_x line. Thus the simplicial elements form a dimensional hierarchy where a p_x simplex of dimension n contains p_x simplices of dimension $n-1$.

This concept of a dimensional hierarchy is not restricted to the faces and edges. Any planar polygon in the 3D reference space can be triangulated into curved triangles, and any line segment in the 2D reference space can be described as a higher order line. However, not all curved regions can be described as a p_x line, triangle, or tetrahedron. Any nonlinearity in the reference space will be compounded in the mapping, and the resulting interpolation will not be p_x.

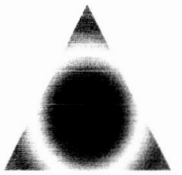

Figure 2: p_2 Shader

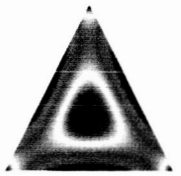

Figure 3: p_3 Shader

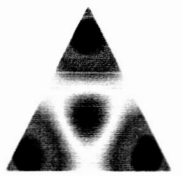

Figure 4: p_4 Shader

4 Shading Parametric Elements

In order to visualize a parametric element with scalar values, s_i, at each node, eq. 2 must be implemented in some manner. OpenGL alone can only do this by refining the triangle or generating a texture map. Both of these methods become extremely slow as the number of triangles increases. An alternative is to use the programmability in the GPU exposed by graphics languages like *Cg*. This is where great performance gains can be obtained. The GPU can inherently use the parallelism in these operations because the rasterization phase generates a pixel at a time (with no dependence on neighboring pixels). The processor can parcel out each pixel in the fragment to the number of raster engines available in the specific graphics hardware.

Eq. 2 can be implemented in a fragment shader by defining texture coordinates at each vertex as the vertex's position in reference space, $\bar{\xi}$, and then evaluating the shape functions in the fragment shader. The results of this shader on one triangle is shown in fig. 2. Figs. 2, 3, and 4 show the results for the p_2, p_3, and p_4 shaders respectively. Note that linear coloring (Gouraud shading) would produce a constant color triangle for each case.

4.1 Performance

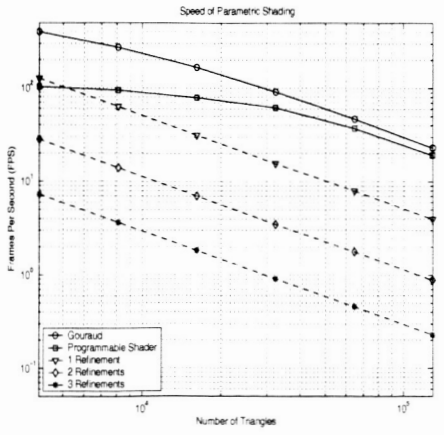

Figure 5: Performance of p_2 Interpolation

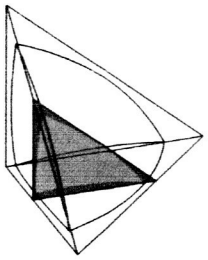

Figure 8: Triangular p_2 Cut w/ Shadow

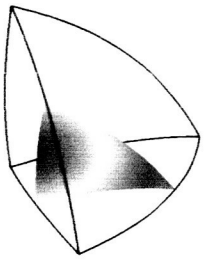

Figure 9: Triangular p_2 Cut Shaded

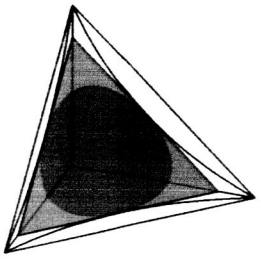

Figure 10: Multiple p_2 Cuts

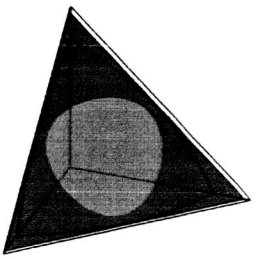

Figure 11: Face Only p_2 Cut

like fig. 10 and fig. 11, a significant portion of the shadow is eventually thrown away. This extra computational burden can be lessened by using eq. 7 to selectively refine the element, and then applying the shadow algorithm to each subelement. As shown in figs. 12 through 14, this *hybrid selective refinement* (HSR) algorithm correctly renders the cutplane intersection while requiring much less refinement than LSR would to produce the same level of accuracy. Also notice that there is some amount of overlap between the shadows, but the reduction in excess fragments more than makes up for this redundancy.

6.5 HSR for 2D data

All elements found in the solution from a 2D flow solver can be thought of as occupying a single plane in 3D space. A shadow that lies in that plane can bound the 2D curved element. This shadow primitive will be a linear triangle C that is congruent to the reduced order triangle R of the element, as shown in fig. 15. This is an extension of the method described in sec. 6.4 where the main difference when visualizing 2D data is in computing the bounds of the element. The maximum value of $p^i(\bar{\xi})$ for a q_2 triangle face always lies at the midpoint.

As with sizing the congruent tetrahedron for a 3D tetrahedral q_3 element, the bounds used for a general 2D triangular q_3 element are looser than those actually necessary for elements used in a computational mesh. The bounds for sizing of δ^i for a general element are:

$$\delta^i = \begin{cases} 1.3p^*_{max} & \text{if } p_{min} < 0,\ p_{max} > 0 \quad \text{(Mixed)} \\ -0.316p_{min} & \text{if } p_{min} < 0,\ p_{max} = 0 \quad \text{(Nonpositive)} \\ 1.125p_{max} & \text{if } p_{min} \geq 0 \quad \text{(Nonnegative)} \end{cases} \tag{30}$$

For a q_3 mesh, assuming that the edge is either concave or convex, using:

$$\delta^i = \begin{cases} 0 & \text{if } p_{min} < 0,\ p_{max} = 0 \quad \text{(Concave)} \\ 1.125p_{max} & \text{if } p_{min} \geq 0 \quad \text{(Convex)} \end{cases} \tag{31}$$

will ensure that C completely covers R.

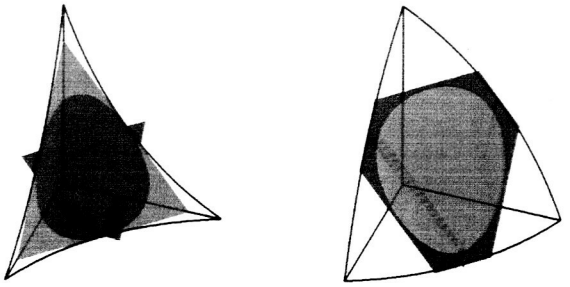

Figure 12: One Hybrid Refinement

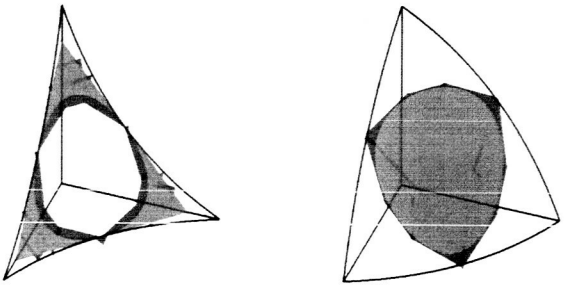

Figure 13: Two Hybrid Refinements

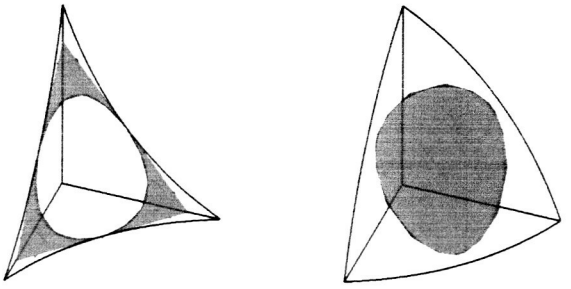

Figure 14: Three Hybrid Refinements

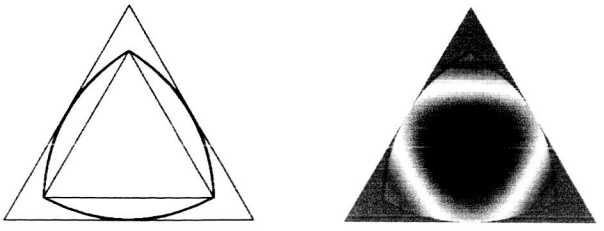

Figure 15: Congruent Shadow Triangle

7 Application to Flow Solutions

The method used to intersect finite elements with planar cuts described in previous sections was developed with the goal of visualizing flow solutions on unstructured grids in both 2D and 3D. This effort supports the work of Project X [4]. The 2D code solves the Euler equations and the Navier-Stokes equations, while the 3D code is currently only inviscid. The equations are discretized using DG methods and solved using p multigrid with line smoothing.

7.1 2D Viscous Navier-Stokes

The approach to solving the Navier-Stokes equations is the same as the method to solve the Euler equations, except that the line smoothing is modified to account for viscous diffusion in addition to convection. The flow around a NACA0012 airfoil at 0° angle of attack was solved using a grid containing 2264 p_1q_1 triangles in the interior and the farfield, and 40 p_1q_3 triangles on the airfoil. Fig. 16 shows the Mach number distribution, which clearly show both the viscous boundary layer and the trailing wake. Fig. 17 shows a close

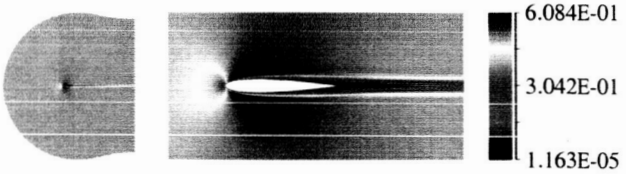

Figure 16: NACA0012 Airfoil Mach Distribution

view of the leading edge, while fig. 18 shows the shadow pixels and outlines the elements. Fig. 19 shows an extreme close-up of just two elements, which are fairly curved. Even at this size, the curvature of the element is preserved.

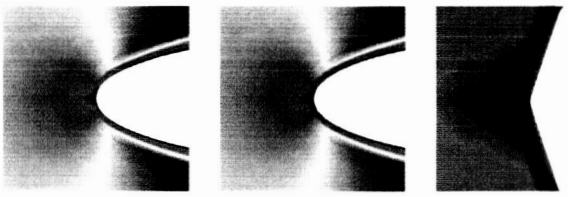

Figure 17: NACA0012 Airfoil Curve

Figure 18: NACA0012 Airfoil Shadows

Figure 19: Two Element Shadows

7.2 3D Inviscid Euler

The application of the 3D code is to a straight NACA0012 wing with a span of 5 chord lengths. The grid used was generated from a 2D airfoil grid, which was then extrapolated into 3D. This produced a tetrahedral mesh consisting of 91936 p_2q_1 interior and farfield elements and 3536 p_2q_3 boundary elements around the wing. The Mach Number distribution is shown along the surface of the wing in fig. 20. Since the grid is fairly well refined around the airfoil, no enhancement was necessary to approximate the shape, though the depth and lighting were modified at each pixel in the fragment program to better approximate the curved shape. The farfield boundary forms a dome around the wing, as seen in fig. 21. Fig. 22 also shows the position of the cutplane.

The vast majority of the elements in the grid are q_1, so the standard marching cubes algorithm handles intersection. However, all the elements that either have a face or an edge on the wing surface are q_3, so that they can accurately conform to the airfoil shape.

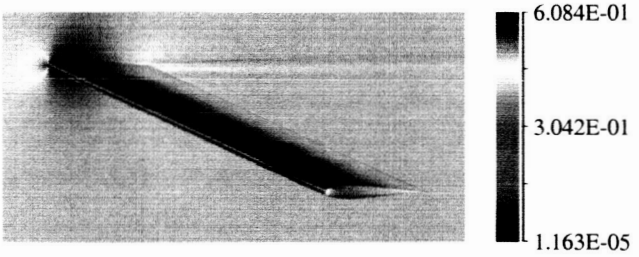

Figure 20: NACA0012 Wing Mach Distribution

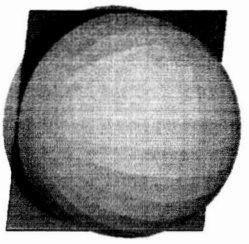

Figure 21: Farfield Boundary

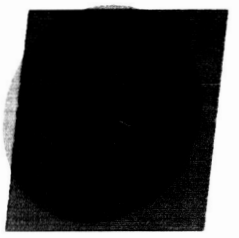

Figure 22: Cutplane Position

A cut through these elements must be rendered using the shadow method of sec. 6, using eq. 27 to generate the shadows. The curvature at the wingtip is best handled with 1 level of selective refinement, so this was used throughout. The cutplane position in fig. 22 was used to generate the following Mach cut in fig. 23:

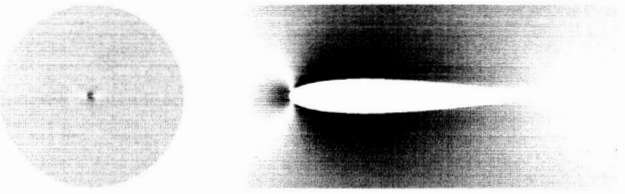

Figure 23: Cutplane Through Mach Field

To provide a better sense of the element size involved, fig. 24 shows the outline of all the q_3 elements that were cut at the position shown in fig. 22. Figs. 25 and 26 show the cutplane through the leading edge, with all the shadow pixels shown in pink. Notice that there is some overlap of the shadow primitives, but since these pixels normally get rejected, this is never noticed by the viewer.

Fig. 27 shows the wingtip, with the cutplane at 3 locations approaching the tip. These cutplane positions were used to generate images through the Mach field and are displayed in fig. 28. This shows that the cutplane shadow method is able to correctly render the planar intersection for even the fairly curved elements at the wingtip.

8 Extension to Isosurfaces

The discussion so far has focused on rendering planar cut intersections, and not on visualizing isosurfaces. The algorithms to render each type of intersection for linear elements are the same, and indeed, the LSR algorithm should work for isosurfaces. The crucial difference is that isosurface will, in almost all cases, be guaranteed not to be planar.

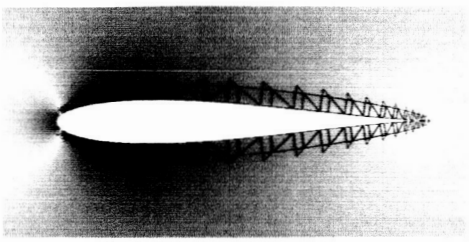

Figure 24: NACA0012 Wing Boundary Elements

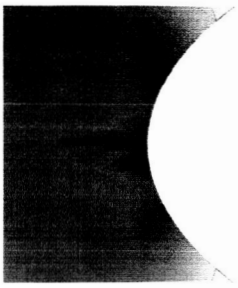

Figure 25: NACA0012 Leading Edge

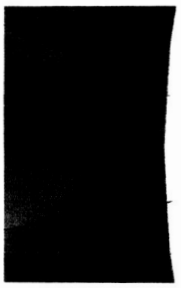

Figure 26: A Few Element Shadows

However, it may be possible to render the isosurface with scalar value, s^*, by bounding it with linear primitives. Based on screen position, x_s, of each pixel on the bounding shadow, the depth is adjusted until the point on the isosurface, x, is found such that x_s lies on top of x (i.e. x and x_s have the same screen coordinates but different depths). To find x, $|s - s^*|$ is first minimized by performing a search of points inside the element that lie beneath x_s, then the fragment can rejected or drawn based on whether or not $s = s^*$. Performing this search would be relatively expensive, so acceptable values of s will lie close to s^* within some bounds set by the accuracy of the search. Under some viewing transforms, the isosurface can curve behind itself, which means there can be multiple solutions, x, that all lie on top of x_s. In this case, the several solutions should be compared using the depth test to determine which one is displayed.

How are the bounding shadow primitives determined to render the isosurface? The faces of the congruent tetrahedron used to generate the cutplane shadow would certainly cover the isosurface intersection, since it captures the entire element by design. But using those triangles could produce many extraneous fragments. This could be alleviated by combining the view-based refinement used in [7] and the selective refinement of HSR to approximate the isosurface intersection.

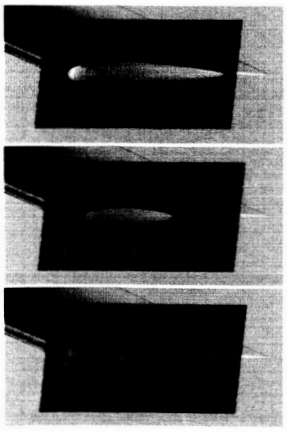

Figure 27: Cutplane Position at Wingtip

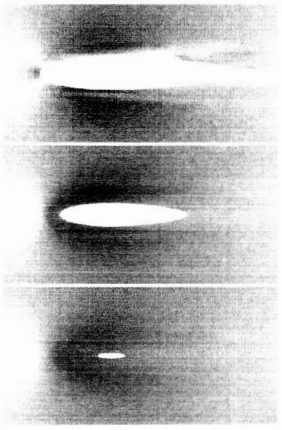

Figure 28: Cutplane Through Mach Field at Wingtip

9 Conclusion

Subdivision algorithms generate exponentially more subelements as the refinement level is increased, and their performance is directly tied to the number of vertices being processed. Programmable shaders leverage the flexibility of modern GPUs to efficiently sample higher order data at each pixel in a powerful manner. Visualizing planar cuts through parametric FEM elements simplifies to knowing the reference coordinates at each pixel, and having the ability to use that information to correctly render the scalar field. The major obstacle is the limitation of having to use planar primitives to generate pixels for the fragment shader. To overcome this challenge, the HSR algorithm bounds the curved intersection with a shadow primitive, which can then be manipulated in the GPU. Some pixels will inevitably be discarded, and to minimize this wasted effort, very coarse selective refinement can be used to generate several shadow primitives that collectively cover the entire intersection. Thus the HSR algorithm provides an efficient and functional method to produce and shade planar cuts through higher order FEM data.

10 Acknowledgements

The work presented here was partially funded by NASA grant NAG8-1872 (Suzanne Dorney, technical monitor).

References

[1] James H. Clark. *A fast algorithm for rendering parametric surfaces.* Computer Science Press, Inc., 1988.

[2] Bernardo Cockburn and Chi-Wang Shu. Runge-kutta discontinuous galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16(3):173–261, September 2001.

[3] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley, Boston, Massachusetts, 2003.

[4] Krzysztof Fidkowski and David Darmofal. Development of a higher order solver for aerodynamic applications. *42nd AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2004-0436*, 2004.

[5] Mike Giles. Personal Correspondence, July 2003.

[6] B. Haasdonk, M. Ohlberger, M. Rumpf, A. Schmidt, and K. Seibert. Multiresolution visualization of higher order adaptive finite element simulations. *Computing*, 70(3):181–204, June 2003.

[7] R. Khardekar and D. Thompson. Rendering higher order finite element surfaces in hardware. *Computer graphics and interactive techniques in Austalasia and South East Asia*, 2003.

[8] Andrea O. Leone, Paola Marzano, and Enrico Gobbetti. Discontinuous finite element visualization. In *CRS4 Bulletin 1998*. CRS4, Center for Advanced Studies, Research, and Development in Sardinia, Cagliari, Italy, 1998.

[9] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proceedings of SIGGRAPH)*, 21(4):163–169, 1987.

[10] P. Solin, K. Segeth, and I. Dolezel. *Higher-Order Finite Element Methods.* CRCPress, 2003.

[11] Luiz Velho. Simple and efficient polygonization of implicit surfaces. *J. Graph. Tools*, 1(2):5–24, 1996.

[12] Luiz Velho, Luiz Henrique de Figueiredo, and Jonas Gomes. A unified approach for hierarchical adaptive tesselation of surfaces. *ACM Trans. Graph.*, 18(4):329–360, 1999.